Why can I not place unmatched braces in Tcl comments

Actually, Tcl has no problem with unmatched braces in comments:

```
puts Hello
# a comment {
puts World
```

This is also just fine:

```
proc greet "
    puts Hello
    # a comment {
    puts World
"
```

The issue people encounter has nothing to do with comments, and everything to do with the fact that Tcl requires that braces be balanced in braced words:

```
proc greet {} {
    puts Hello
    # a comment {
    puts World
}
```

The third argument to proc is only a block of code in the eyes of proc. The Tcl interpreter must first parse the command into words in order to know what arguments to invoke proc with, and according to the rules of Tcl, "braces nest within the word".

In the following example of a proc command with nested braces, the # character in the third argument is not special. It's just a literal # character:

```
proc foo args {
    puts "executing procedure foo"
    # unsuccesfully attempt to comment out this code block {
        foreach a $args {
            puts "argument: $a"
        }
    }
}
```

Although this third argument is a well-formed string, it's not a syntactically-valid Tcl script. Later on, when foo is called, an error is raised:

```
$ foo hello world
argument: hello
argument: world
invalid command name "}"
```

Tcl recognized # as a comment and ignored ignored the rest of that line, *including the trailing open-brace*. After successfully evaluating foreach, the interpreter came upon a line containing a single character, }, tried to find a corresponding routine for that name, and came up empty.

Commenting out Blocks of Code

You may have been trying to comment out a block of code, like this

```
# comment out this block {
    proc foo {} {
        blah ; blah ; blah
    }
}
```

and found that the comment ends at the end of the line, paying no special attention to the brace on that line.

One way to command out several lines of code is to prefix them all with #. Another alternative is if 0 ...:

```
# The second argument is not evaluated:
if 0 {
    proc foo {} {
         blah ; blah ; blah
    }
}
```

Another alternative is to use a command that does nothing with its arguments:

```
proc comment args {}
comment {
    proc foo {} {
        blah ; blah ; blah
    }
}
```

A slightly sillier way to avoid evaluation is to make the script part of a procedure::

```
#  comment out this block using 'proc'
proc donteverrunme {} {
    proc foo {} {
         blah ; blah ; blah
    }
}
```

(KJN *Rule 9 on the Tcl man page [L1] deals with comments. The key words in that rule are "where Tcl is expecting the first character of the first word of a command". When (e.g.) the proc command is evaluated, a comment line in the proc body does* not *have*

that status. This aspect of Tcl has confounded me many times. I wish there were a better Tclish way to handle comments. Also, while the 11 rules of the man page are short and sweet, they must be read with the precision of a language lawyer.)

These rules are relatively easy to follow in isolation but the problems come about when they are nested.

Take for instance the following code.

```
proc a {a b} {
    if {$a < 0} {
        puts "Negative \{"
    } elseif {$a > 0} {
        puts "Positive \}"
    } else {
        puts "Zero {}"
    }
}
```

When the **proc** command is evaluated the parser has split its arguments into the following words (the | are delimiters and as such are not part of the words).

```
1: |a|
2: |a b|
3: |
    if {$a < 0} {
        puts "Negative \{"
    } elseif {$a > 0} {
        puts "Positive \}"
    } else {
        puts "Zero {}"
    }
|
```

The parser did not look inside word 3 because that is up to the command and so it only used rule 2.

(*) The byte compiler does invalidate this slightly as it 'knows' how to parse certain commands and therefore will parse before evaluation.

switch

Watch out for comments inside **switch** commands, since the switch command does not itself interpret # as anything other than an ordinary character. Thus, you must make sure that they are only ever put *inside the body* parts, as otherwise you will end up with either weird parsing or an out-and-out error.

Thus, the following code is wrong:

```
switch $foobar {
    # Match a first
    a { puts "Matched a" }
    # Match b second
    b { puts "Matched b" }
}
```

This is because it is equivalent to *(i.e. precisely the same as)*:

```
switch $foobar {
    # Match
    a first
    a { puts "Matched a" }
    # Match
    b second
    b { puts "Matched b" }
}
```

Which is the same as:

```
switch $foobar {
    # Match
    a first
    b second
}
```

Not what was intended! The following is possibly even worse (depending on your point of view...)

```
switch $foobar {
    #Match a first
    a { puts "Matched a" }
    # Match b second
    b { puts "Matched b" }
}
```

This is equivalent to:

```
switch $foobar {
    #Match a
    first a
    { puts "Matched a" } #
    Match b
    second b
    { puts "Matched b" }
}
```

In this case, the body of the switch has an unpaired word, and so the switch command moans conspicuously. The *correct* way of writing the above is:

```
switch $foobar {
    a {
        # Match a first
        puts "Matched a"
    }
    b {
        # Match b second
        puts "Matched b"
    }
}
```

DKF:

What other surprises do we want to document in this area?

Well, here's another case with similar troubles -- JC

```
array set config {
    # this comment is messing up things ...
    path /usr/local/bin
    user admin
}
```

Quoth an anonymous person via firewall.cmsis.com...
This problem could be removed from the language completely by modifying the lexical analyzer portion of your interpreter to simply skip over any characters from a "#" to a "\n". However this could affect existing code you have, by forcing you to backslash escape the "#" character in places where you didnt have to.

DKF replies...
This is actually very difficult to do, since Tcl doesn't have conventional lexical analyser and parser stages (which has tremendous advantages in terms of lignuistic flexibility.) Comments are defined to act by causing the interpreter to ignore all characters through the next new-line. The thing is that comments only start in places where the interpreter is expecting a command to start, and the interpreter puts off making that decision for as long as possible. If you feel this is easy to fix (despite all the people telling you that it isn't) then perhaps you should have a go at "fixing" it to your satisfaction. The source code is freely available, so you can make whatever changes you want. If you can figure out a way of doing it that doesn't break thousands of scripts, we'll talk...

LV replies...
One thing that often is not thought about when someone starts with **it should be easy** is that tcl treats data and commands identically. One is not required to surround data with quotes, etc. Thus, when the lex/parse code hits a #, all it can tell is **is this the beginning of a command**. It doesn't know things like **am I in the middle of data**.

JC argues...

I understand quoting (I wrote TinyTcl, so I better!), but the original poster does have a point. Just like it is possible for the scanner to replace "\<newline><whitespace>" by a single space (everywhere), it could replace "#<anything><newline>" by <newline> (everywhere), unless the hash was preceded by a backslash. It would also affect list-to-string conversions. This can break scripts, and it can even break data-as-script, but it *is* a trade-off - a weekly thread on c.l.p versus a gotcha for everyone upgrading to the latest Tcl release.

Bryan Oakley points out...
...that the "break data-as-script" argument is a pretty darn big thing to break. It is for this reason more than any other that we absolutely *can't* preprocess comments and throw them away. It's just too hard to know what is a comment and what is data. For example, consider the following fragment. Is the data following the # a comment that should be thrown away? There is no way for the interpreter to know, and I sure as heck don't want it to guess:

```
set foo {
    # is this a comment?
}
```

PSE notes...: I wish Scriptics was as conscientious about breaking *thousands of Tcl scripts* as you folks are!

LV: Scriptics is that conscientious - or they would have changed this as well as several other things that people report as bugs dozens of times every day. I haven't had to change a Tcl script due to Tcl changing the script level language in about 5-7 years (whenever the change relating to geometry vs width and height occured).

LV: In news: I found a variety of interesting comments relevant to this page:

Bob Techentin wrote:

```
 >
 >   If a hash character (`#`) appears at a point
 >   where Tcl is expecting the first character of the
 >   first word of a command, then the hash character and
 >   the characters that follow it, up until the next
 >   newline, are treated as a comment and ignored.
           ^
 not preceded by a continuation character, or end of script

 > The
 >   comment character only has significance when it appears
 >   at the beginning of a command.
 >
```

The above perfectly describes the behaviour of the Tcl comment, however it is too concise and relies on the reader to completely understand the rest of the man page before they can really understand this. I wish that Scriptics would improve the

documentation to at least explain why it is the case and why it is unlikely to change.

Also, it is possible to provide much more useful error messages. The indentation of the different blocks can be used to decide what the user probably meant, if a } is found whose indentation level is different from the line on which its matching { was found then there could be an unbalanced { somewhere. Similarly while parsing blocks Tcl could looks for comment like structures and possibly strings containing {}.

If an error occurs, note that I am not using this information unless an error occurs, then Tcl would provide this extra information to the user which may help them track down their problem.

How about adding something like this to the documentation, in a seperate man page *(rendered for Wiki between the next two horizontal rules by DKF)*

Consequences of Tcl's quoting rules

Note: Tcl cannot distinguish between data and code until it is actually asked to execute a block. It is only at this point that it can determine what the commands are and hence what the comments are.

Prior to being asked to execute a block Tcl's parsing is limited to counting unquoted {}s if the block starts with a {, or finding the next unquoted " if the block starts with a ".

This means that it is not possible to place unbalanced and unquoted {}s anywhere in a {} delimited block, or unquoted "s in a " delimited block. Anywhere includes comments and strings.

The behaviour of switch is a prime example, the body of the switch which contains all of the different cases to check is not code, it is data to the switch command, switch decides what to do with the data itself and switch does not support comments in its data. The body of each case is a block of code and as such can contain comments.

It is understood that the behaviour of the Tcl comment may be strange to people used to other languages, hence the reason for this lengthy explanation and the following examples. However, it is not possible to provide a completely backward compatible preprocessing step which can make comments behave as you may expect. This is because it is not possible in general to tell just from looking at some Tcl code which blocks are data and which are code because Tcl treats all commands exactly alike.

As you can see the comment behaviour is an unfortunate side-effect of the structure of the Tcl language in much the same way as the Tcl programming style of

```
if {...} {
    :
}
```

rather than

```
if {...} \
{
    :
}
```

is. You will find that you will be much more productive in Tcl if you modify your expectations and style to suit it.

If you need or want to use a preprocessing step of your own then it is very easy to write your own in Tcl. However, because it is not going to be completely backward compatible do not expect that everyone will want to use your version.

Examples

```
# This is a comment at global scope, note no enclosing
# braces.

# An unbalanced { in a comment works at global scope
# because it is not surrounded by {}.

set a "A string containing an unbalanced {"

                v
proc example1 {} {
     # However place an unbalanced { in here and you
     # will cause problems because it means that when
     # Tcl parses the block starting at "v" the block
     # will not be terminated by the brace at "^".
}
^

                v
proc example2 {} {
    # A solitary { in one comment.

    if { ... } {
    }

    # Followed by a } in another comment inside
    # the same block does not cause a problem because
    # the block starting at "v" ends at "^" as you
    # expect.
}
^

proc example3 {} {
  # Unbalanced {
#        v
  if {1} {
    # Balancing }
  }
# ^
# The outer block is balanced but when it is
# executed the parser strips out the comment on the
# first line ignoring the { and so when it comes
# to parse the block starting at "v" it ends it
# at the first } it comes across so the one at "^"
# is assumed to be the name of a command and as
# no such command exists an error is generated.
}

                v
proc example4 {} {
    set "this problem is not really a comment problem because
if I put an unbalanced { in here then I have exactly the same
problems"
}
^

                v
```

```
proc example5 {} "
    # If I change the delimiters from {} to double quotes then
    # I can have as many {{{{ and }}}}}}}} in here as I like
    # and it will not cause a problem because Tcl's parser is
    # not counting them any more, it is looking for an unquo ted
    # \". It is not recommended that you use \"s as delimiters
    # to procedures as Tcl will perform substitution on the body
    # which may cause some problems if you are not prepared for
    # it.
"
^
```

Stephen D. Cohen (SDC) Asks:

What I do not really understand is why # is afforded any special treatment at all. Why is it not simply a command that takes a variable number of arguments and does nothing? That does not seem to be too far from what it does now, and does not appear as if it would break any of the above. Then something like:

```
proc foo { mire } {
# This is a line that comments the rest of this muck... {
    set num [binary scan $mire H* muck]
    puts "The muck was: $muck"
}
}
```

Would do the nothing that I (and most reasonable people?) would expect. In data, the # would have the same effect it has now - basically nothing unless special quoting / extreme measures are taken.

I guess I am asking: Why has "#" been endowed with special powers that only seem to make it behave in a manner that is inconsistent with the rest of Tcl? Or am I missing something either deep / philosophical or intuitively obvious to the casual observer?

Am I going to wish I never screwed up the courage to post on the Wiki? :-)

An unidentified poster responded:

the answer may be that without the special treatment, you can't have comments with unbalanced quotation characters in them. With the special behavior you can. In practice, the current behavior works 99.9% of the time, so why change?

and SDC Responds:

So the answer may be that since this has been a problem since the beginning of time, it must remain a problem. Such thinking is sure to enocurage progress.

I would also argue that the current behavior does not work 99.9% of the time, unless the developer keeps the special rules for # in mind. Why then, I ask, are we using up precious programmer creativity on a command which does nothing?

I am not spoiling for a fight or anything, I am just looking for a good answer to why # is not simply a command that does nothing. One of the Tcl insiders must know...

Another unidentified poster responded (SDC cleaned this up so it prints correctly)

is not treated as another command for at least this reason:

```
% set a try
try
% # [puts $a]
% set b [puts $a]
try
% puts $b
```

After the #, tcl code isn't executed. If it were treated like any other command, that would no longer be the case - so one wouldn't REALLY be commenting out lines...

To which SDC Responds:

OK, that certainly makes it obvious why # has to be endowed with *some* special powers. :-) I did not really mean that # should just be a command with *no* special powers, just that its parsing should be the same as every other command. I thought it went without saying that the arguments to # should not be evaluated.

I am still looking for the answer as to why the parsing of the # command is not the same as the parsing for every other command in Tcl, with the exception that parameters to # are not evaluated at all.

Lars H: There are a few character combinations which cannot be parsed as commands at all, and it is useful that comments can contain them. Consider

```
# Some ASCII graphics:  `{}{}=={}{}`
# Some C:  Tcl_DStringAppendElement(&cmd, (newName ? newName : ""));
```

Both of these are OK in comments, but neither can be part of a command:

```
% list Some ASCII graphics:  {}{}=={}{}
extra characters after close-brace
% list Some C:  Tcl_DStringAppendElement(&cmd, (newName ? newName : ""));
extra characters after close-quote
```

Adding e.g. $a (if there is no variable a) at the beginning of these commands will still produce the same error messages, thus confirming that these errors are detected before Tcl has done any substitutions.

As far as I am aware, there are no special exceptions for any command parsed.

Ingemar Hansson: Wouldn't also the comment

```
#Comment without white space after the # sign
```

break SDC's suggestion? In the above example the command - as SDC sees it - isn't # anymore, but #Comment.

FW: Yeah, it would, but as an alternative it could automatically capture all commands starting with #. But anyway, we've alredy proved SDC's idea unusable, so it's a moot point.

NEM: I'm not entirely sure any of the arguments against having comments as commands are particularly convincing. Which is harder to explain: requiring people balance braces in comments, or requiring people properly quote comment strings, and put a space first? For instance, lots of people seem to do:

```
if {0} { ... }
```

to comment out blocks of code. Why not simply make that:

```
# { ... }
```

? Tcl won't touch anything between the braces anyway, and it would allow you to do cool things by redefining the # command to do e.g., automatic documentation generation, meta-data etc. I can see it would probably be confusing to people used to other languages that comments are just commands too, but I don't see that as a particularly huge problem. Still, probably too late to do anything about this now, as it would definitely break lots of scripts (and changing the behaviour of comments would be a particularly nasty way to break someone's script!). One more serious breakage of this scheme would be #! lines at the start (IIRC, some systems don't like a space between #! and the path). Ah well, perhaps best to leave # as it is, and write a new command for comments (like XOTcl's @ stuff).

RS: Such a "block comment" is easily implemented, if you just require one quoted argument, as e.g.

```
 interp alias {} @ {} if 0
```

:^)

GN: Another advantage of using a command like @ is that it can be easily overloaded. The idea of the XOTcl commenting stuff is to provide "RDF-style" meta-data including comments to arbitrary parts of a program (files, procs, objects, classes, methods, arguments...).

See also Why can I not start a new line before a brace group.

We need a standard one-paragraph response to the many people who regard Tcl's commenting as "brain-damaged" on their first encounter. Maybe this section should be near the top of the document.

There certainly are plenty of such people.

Lars H, 2006-08-04: Here's an exercise (possibly a braintwisters candidate) for people who think Tcl should ignore braces after hashes in scripts when looking for the brace ending a script argument.

```
proc X {args} {set args}
X {
X list
# } [
list }
proc X {script args} {lappend args [eval $script]}
X list ]
```

It turns out that the first argument of the first X command will be treated as a script if and only if one does not let the # "hide" the first right brace from the balancing of braces, i.e., a parser that tries to guess whether words of a command are scripts or not and uses that guess to give # precedence over } or not will always get it wrong.

AMG: Indeed yes, and thank you for this example. Giving # the power to inhibit brace counting until the next (non-backslashed) newline would have to applied consistently with no heuristics to decide whether any given open brace starts a script or not. Naturally, treating every braced string as a Tcl script causes trouble when the string is actually some other language for which # means something other than a line comment. Example:

```
puts $file {#include <stdio.h>}
```

This works great in current Tcl but fails if # inhibits brace counting. It becomes impossible to use braces to quote any string containing # not followed by a newline (itself not immediately preceded by zero or an even number of backslashes). In such cases, backslash or double quoting must be used.

```
puts $file #include\ <stdio.h>
puts $file "#include <stdio.h>"
```

Okay fine, suppose we accept this negative consequence. Now consider one-liner procs:

```
proc x {file} {puts $file #include\ <stdio.h>}
proc y {file} {puts $file "#include <stdio.h>"}
```

Uh oh, should these be banned too? Or should #'s brace inhibition powers themselves be inhibited when # is not in a position where it could start a comment?

Very quickly, brace counting rules went from "match braces except those preceded by an odd number of backslashes" to "match braces except when punching the price of tea in China into a calculator and turning it upside down produces a humorous display." Not

impossible, and from a basic user perpective less surprising than the fact that braces match even when seemingly inside comments. But this is getting really hard to codify.

And if you think it's hard enough for the parser to decide whether or not to count any given brace toward matching, remember that the code for generating the string representation of a list also has to know the rules inside and out so that it can correctly quote any given word such that a command prefix can be stored in a list and its string representation later eval'ed.

What decisions must the string representation generator make when the list is one of the above proc definitions? Ideally, it would recognize that the final element could be simply brace-quoted due to the # not being in a position to inhibit counting of the close brace. That means it must be given parser-like powers to recognize which semicolons and which newlines are and are not command separators, as well as which double quotes come at the start of a word and therefore can quote away the power of # to comment away the power of braces. It gets worse: it must also be able to stack its evaluation state whenever it hits square brackets since they contain a fresh script even when inside double quotes. Then one final twist of the knife: in current Tcl, # inhibits close brackets!

Like in the case of Lars H's example, it's very easy to imagine that a set of rules complicated enough to satisfy all of the above would contain some fatal but deeply hidden contradiction.

AMG: Related:

```
proc moo {x} {
    if {$x} {
        puts "{"
    } else {
        puts "}"
    }
}
```

Try this and see what happens. [moo 0] does nothing, and [moo 1] prints "{" then errors "extra characters after close-brace". If you've read the rest of this page, you can understand why.

The fix is to backslash braces that are not intended to count toward brace counting.

DKF: Compare also this:

```
proc moo2 {x} {if {$x} puts\ "{" else puts\ "}"}
```

That *doesn't* have the problem. It's the interaction of the inner braces which actually has the difficulties. Let's show this like this:

```
% lindex [info body moo] 2

        puts "{"
    } else {
        puts "}"

%
```

What's happened? Easy: the " characters haven't been considered *at all* when counting the braces. Fortunately, this is not too big a problem most of the time.

---

apl - 2018-03-16 11:42:42

This is also just fine:

```
proc greet "
    puts Hello
    # a comment {
    puts World
"
```

you mean:

```
set greet "
    puts Hello
    # a comment {
    puts World
"
puts $greet
```

?

---

apl - 2018-03-16 12:59:39

... or rather:

```
proc e {} "
    puts Hello
    # a comment {
    puts World
"
```